NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-85-150

Escher--A Geometrical Layout System For Recursively Defined Circuits

Edmund Clarke, Yulin Feng

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

July 1985

This research was supported by NSF. Grant Number MCS-82-16706

ABSTRACT: An Escher circuit description is a hierarchical structure composed of cells, wires, connectors between wires, and pins that connect wires to cells. Cells may correspond to primitive circuit elements, or they may be defined in terms of lower level subcells. Unlike other geometrical layout systems, a subcell may be instance of the cell being defined. When such a recursive cell definition is instantiated, the recursion is unwound in a manner reminiscent of the procedure call copy rule in Algol-like programming languages. Cell specifications may have parameters that are used to control the unwinding of recursive cells and to provide for cell families with varying numbers of pins and other internal components. We illustrate how the Escher layout system might be used with several nontrivial examples, including a parallel sorting network and a FFT implementation. We also briefly describe the unwinding algorithm.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

ESCHER--A Geometrical Layout System For Recursively Defined Circuits

Edmund Clarke, Yulin Feng

1. Introduction

Many circuits such as sorting networks, hardware multipliers, and FFT implementations can be described by recursive geometrical patterns. Some layout languages provide support for recursion ([2], [6], [7] and [10]); however, in all such systems familiar to us the circuit description is textual rather than geometrical. We believe that it is more natural to describe complicated circuits geometrically, rather than by giving a textual description and requiring that a program figure out the details of the layout. Some circuit editors have powerful iteration operators that can be viewed as implementing a form of tail recursion [3], but none allow full recursion. We have implemented a geometrical layout system (called the Escher System) in which recursive patterns can be specified directly and then instantiated to obtain layouts for complex circuits automatically.

An Escher circuit description is a hierarchical structure in which the basic building blocks are cells, wires, connectors between wires, and pins that connect wires to cells. Cells may correspond to primitive circuit elements such as NAND gates and latches, or they may be defined in terms of lower level subcells, which are defined in terms of even lower level subcells, etc. By using the Escher system, a number of primitive cells can be connected together in complex geometrical pattern to describe the layout for a large and intricate circuit. Designers do not need to worry about the absolute sizes and positions of various circuit components; only the topological relationships are important. Moreover, the system is completely interactive. Circuit diagrams are constructed using a pointing device ("mouse") and tablet.

Although many circuit editors provide a set of features similar to the ones that we have just listed, our system is unique in that a subcell may, in fact, be instance of the cell being defined. When a recursive cell definition is instantiated, the recursion is unwound in a manner reminiscent of the procedure call copy rule in Algol-like programming languages. Cell specifications may have non-negative integer parameters that are

used to control the unwinding of recursive cells and to provide for cell families with varying numbers of pins and other internal components. While the notion of parameterized cell specifications is quite common in textual hardware description languages, we believe that it has not been previously used with graphical circuit editors and, therefore, may be of independent interest.

Our paper is organized as follows: Section 2 describes the various notational conventions that the Escher system uses for specifying recursive circuits. Since recursive cells are usually parameterized by some integer variable, special conventions are needed for describing groups of subcells that depend on the parameter. In section 3 we give two examples of how the Escher System might be used with recursive circuits that are based on parallel *divide and conquer* strategies. We believe that the Escher system will prove most useful for laying out circuits with this type of structure. Section 4 shows how the Escher System might be used for laying out a more complicated example, the parallel prefix circuit originally described by Fisher and Ladner ([1]). In sections 5 and 6 we discuss how the Escher system works. Section 5 briefly describes how various circuit components are represented in the system. This section also addresses the question of how much circuit components may be moved around in obtaining a layout. The algorithm that unwinds and lays out a recursive diagram is outlined in section 6. Since basic subcells must occupy a fixed area, the algorithm must proceed bottom up, expanding each higher level cell so that all of its lower level subcells will fit. The paper concludes in section 7 with summary and discussion of ways in which the Escher system might be extended to produce better layouts.

2. Conventions for Specifying Recursive Circuit Diagrams

As an example of how the Escher system might be used, we consider the problem of laying out the Tally circuit described in [8] and also in [9]. This circuit has n inputs and $n+1$ outputs. The k-th output will be high and all other outputs low, if exactly k of the inputs are high. Figure 2-1 gives the Escher version of a recursive definition for the Tally circuit.

In the specification there are two kinds of cells: *Basic* cells that cannot be refined further (like the two input multiplexers), and *Composite* cells that contain other cells, wires, and connectors (like the recursive occurrence of Tally(n-1)). The cells that are directly contained within a composite cell are its *subcells.* Sometimes several subcells $S_1, S_2, \ldots, S_n$ are instances of the same cell C. In this case we say that C is the *source* of each of the S_i's.

Since the specification is parameterized by n, some abbreviations are needed to represent groups of lines and subcells that depend on n. When a definite value is provided for n, each such abbreviation in the specification may be evaluated.

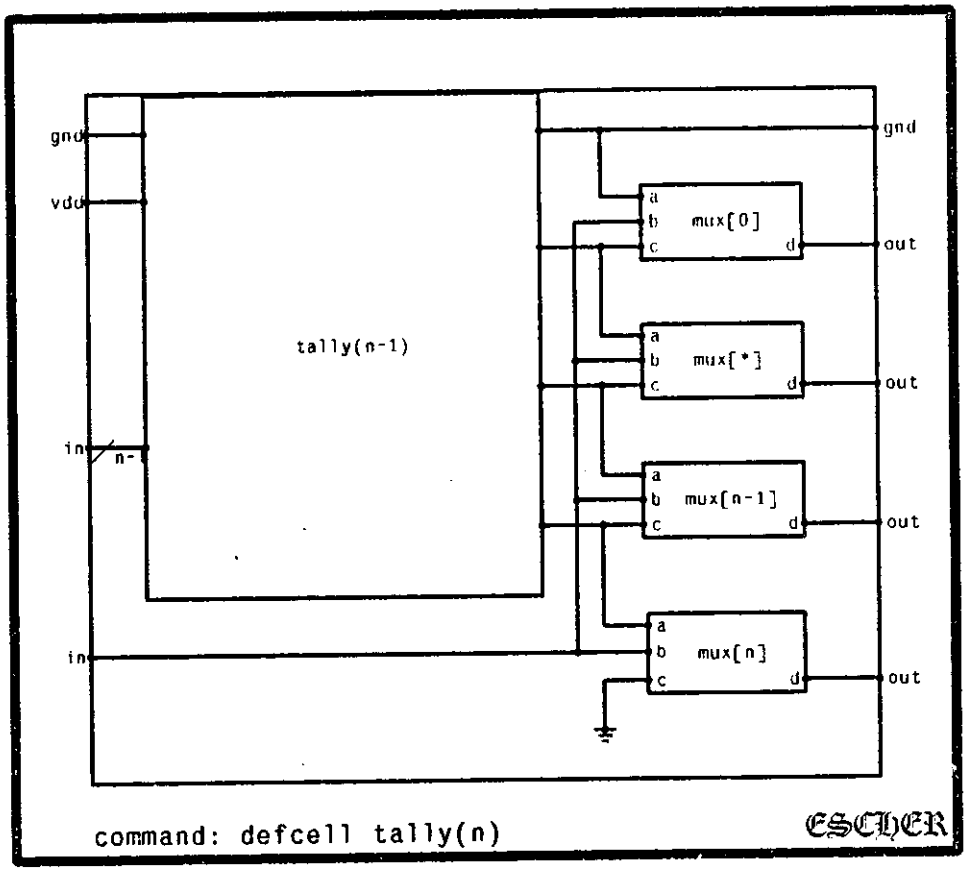

Figure 2-1: Recursive Pattern for Tally(n)

Groups in Escher are somewhat like one dimensional arrays in programming languages. A *group* is a horizontal or vertical array of identical cells with the appropriate interconnecting wires. The subcells of a group may be either basic cells or composite cells. They are distinguished from one another by an integer index, which increases from left to right in the case of a horizontal array or from top to bottom in the case of an vertical array. The initial and final values of the index may depend on a parameter of the cell containing the group; however, the increment must be a fixed positive integer. A group whose length depends on an undetermined parameter is represented by three subcells, one for the first subcell, one for the last subcell, and one in the middle with index "*" to represent all of the remaining subcells. Thus, the "*" serves exactly the same function in our formal specification that the ellipsis "..." serves in an informal specification. A number appearing after the "*" represents an index increment; when the "*" appears alone, the default value for the increment is 1. In the Tally example (Figure 2-1) there are a total of (n+1) multiplexers, the MUX[n] and

MUX's with indices from 0 to n-1 in a group. When a group of subcells is specified, it is only necessary to give the the position of the first and the last subcells of the group with respect to some other part of the circuit. When the containing cell is instantiated and all of the parameters of the group are fixed, this information is sufficient to determine the position of each of the subcells of the group.

Finally, Escher uses a short diagonal mark on a wire to represent a group of wires. An expression associated with the mark indicates how many lines are in the group. We call such groups of wires, *buses*, and the associated expression, the *bus width*. In Figure 2-1 there are two buses, and each represents n-1 wires. We also use the convention that a wire connected to a subcell with index "*" actually represents the same number of wires as the number of omitted subcells.

The circuit for the base case, Tally(1), is shown in Figure 2-2.

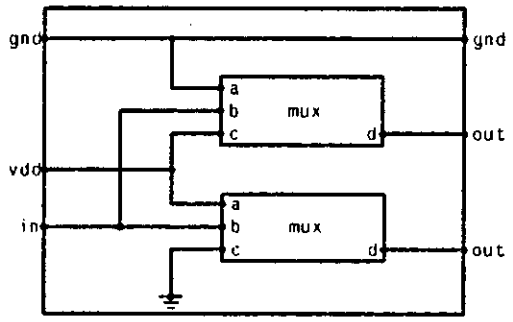

Figure 2-2: TALLY(1), a base case for TALLY recursive specification

Examination of the recursive specification for the TALLY circuit immediately shows how it works. Each multiplexer has three inputs labeled *a*, *b*, *c* and one output labeled *d*. If *b* is high, the output *d* selects the value *c*; otherwise, it selects the value *a*. It is easy to see that the base case is correct. We assume that TALLY(n-1) is correct and that k of the first n-1 inputs are high. By the induction hypothesis, the k-th output of TALLY(n-1) is high. If the n^{th} input is also high, then all of the selector inputs of the multiplexers will be high, so each MUX's with index in the range from 0 to n-1 will select as its output the value of its *c* input, while the output of MUX[n] will be low. Thus, the $(k+1)^{th}$ output (counting from bottom to top) of TALLY(n) will be high and the other outputs will be low. A similar discussion can be used for the case in which the n^{th} input of TALLY(n) is low.

After we instantiate the TALLY circuit with a given value, for example, n=6, the Escher system will

automatically unwind the recursive specification into the circuit diagram shown in Figure 2-3. A final phase (that has not been completed) will compact the circuit diagram produced by the Escher system in accordance with a set of design rules appropriate to the transistor technology used to fabricate the chip.

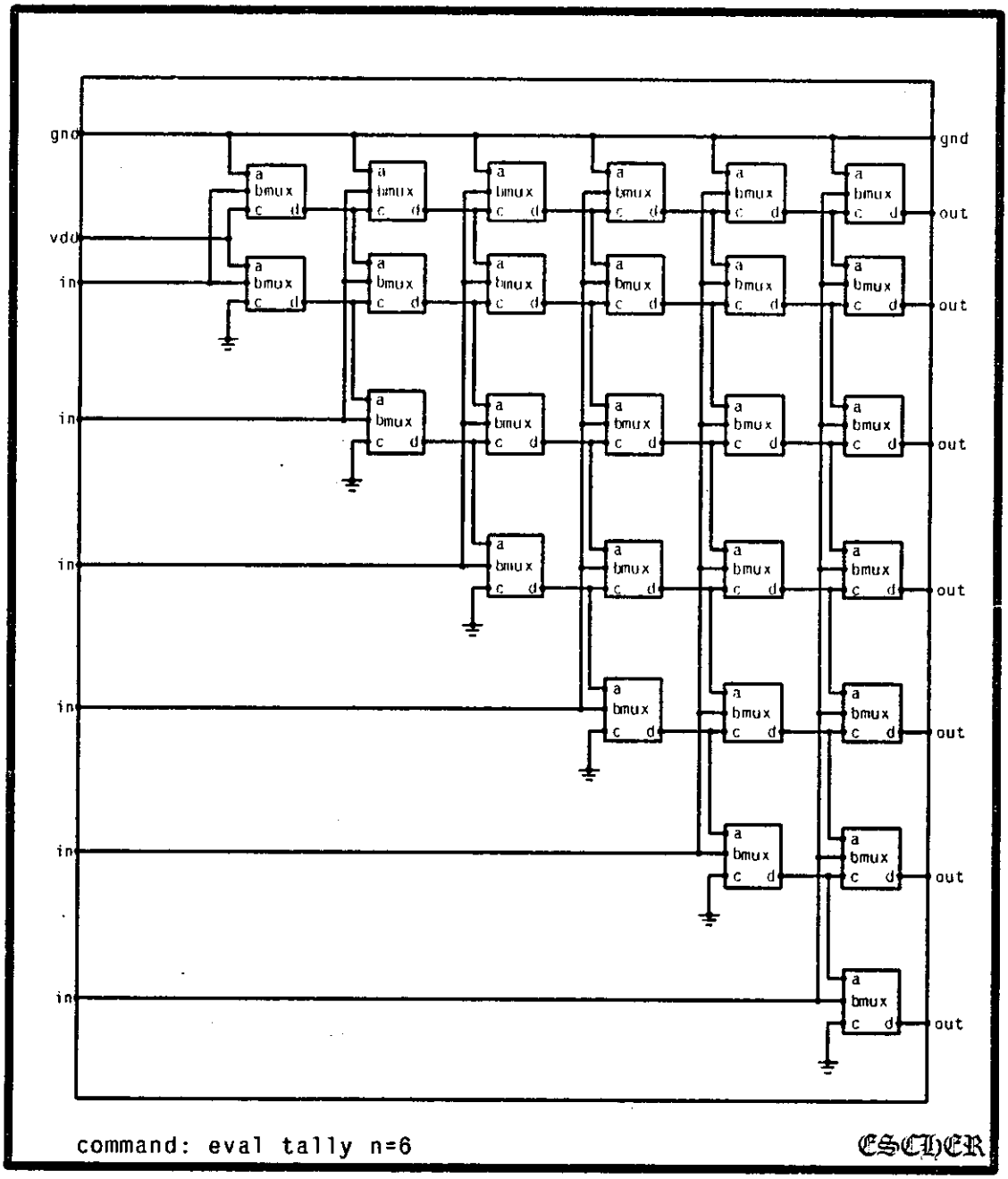

Figure 2-3: Instantiation for Tally(6)

3. Divide and Conquer Circuits

The simplest recursive circuits have only a single recursive sub-circuit. This case is somewhat like tail-recursion in programming languages and is relatively easy to implement. The Tally circuit in Figure 2-1 is an example of such a recursion. Unfortunately, not all recursive circuits have such a simple structure. Many interesting circuits are based on a *divide and conquer* strategy in which a complicated task is realized by a number of subcircuits each of which is a recursive instance of the circuit being defined. Adders, multipliers, sorters, FFT circuits, etc., can all be structured in this manner. Figuring out by hand an appropriate layout for an instance of such a circuit can be quite tricky. Once the recursive structure of the circuit has been determined, the Escher system may be used to unwind a particular instance of the circuit. We illustrate below how Escher might be used with two well known examples of recursive divide and conquer circuits.

3.1. Example 1: Parallel Sorting

Our first example is a network for sorting a sequence of n k bit numbers into increasing order, where n is assumed a power of 2 [5]. The standard divide and conquer approach is to sort the first half and the second half in parallel and then merge the two sorted sequences. The Escher specification for such a circuit is shown in Figure 3-1. Note that every bus width number here means the number of k-bit wires.

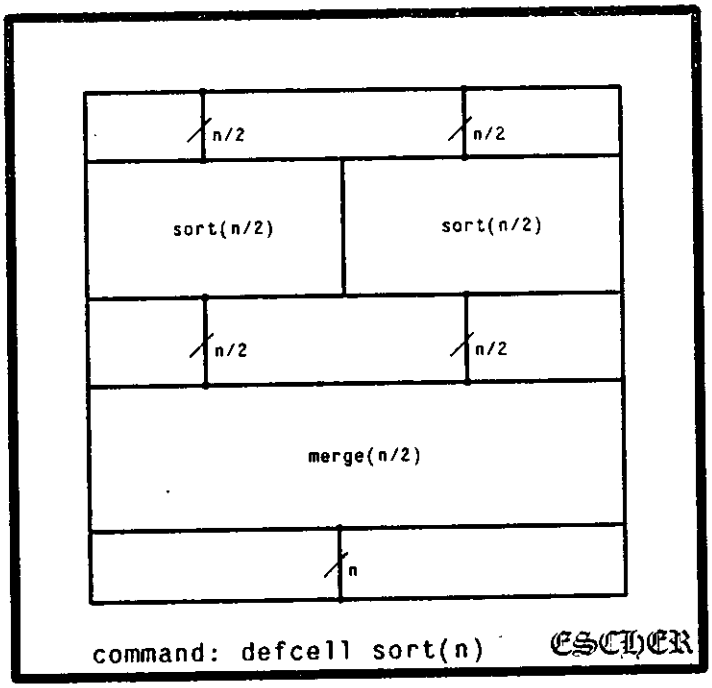

Figure 3-1: Recursive pattern for Sort(n)

The Merge cell can also be defined recursively. To merge two sequences "a" and "b", we merge the even-indexed elements of "a" with the odd-indexed elements of "b", and the odd-indexed elements of "a" with the even-indexed elements of "b". The outputs of the two half-size merging circuits are sent through an array of comparators. Each comparator "CMP" sorts two k-bit numbers in order. Figure 3-2 gives the recursive definition of Merge(n). Pass(n), shown in Figure 3-3, contains only wires and is used to separate the even-indexed inputs and the odd-indexed inputs.

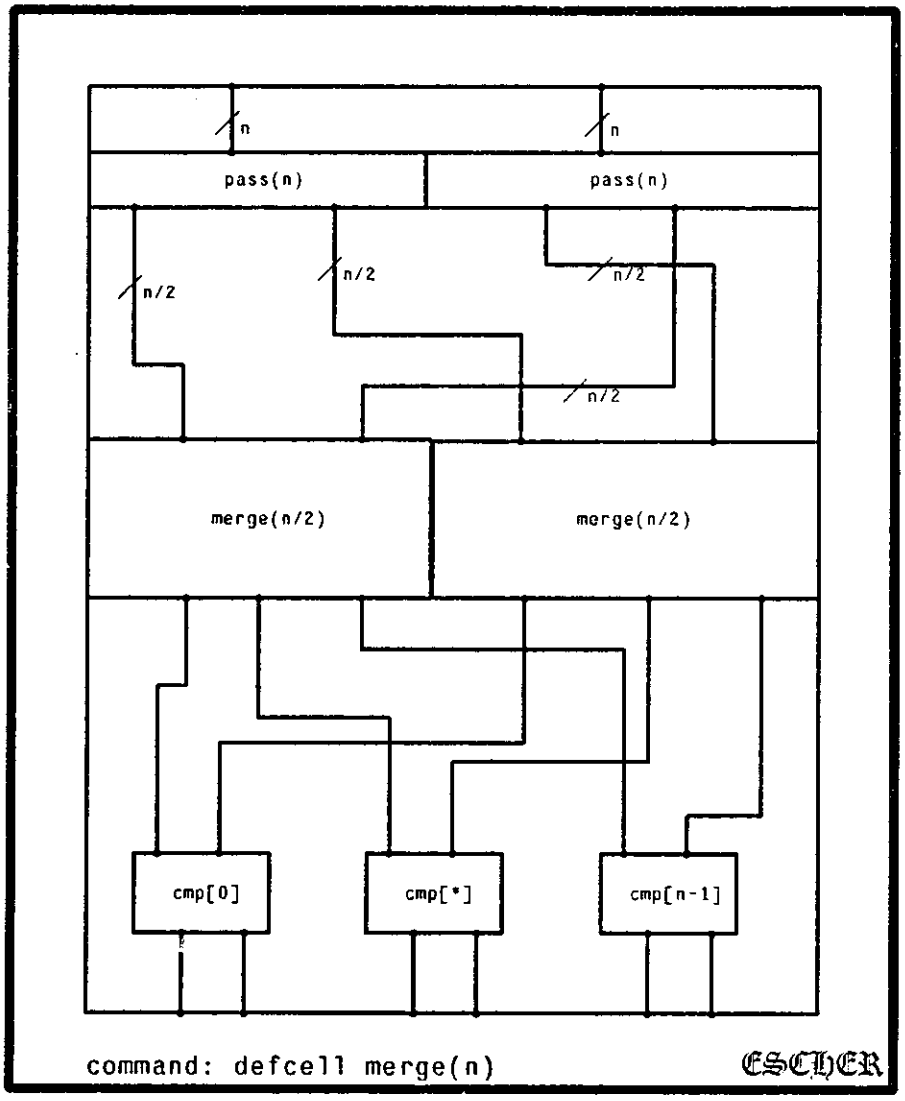

Figure 3-2: Recursive pattern for Merge(n)

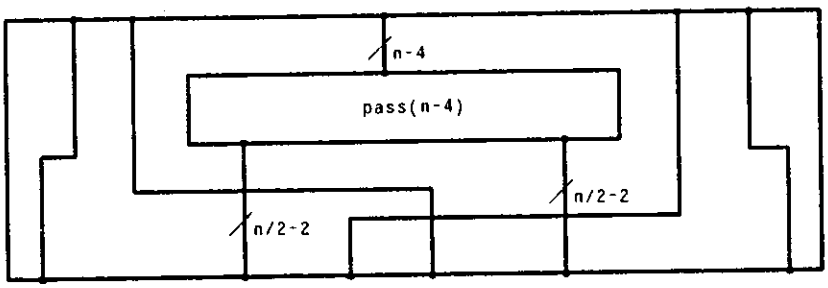

Figure 3-3: Recursive pattern for connections Pass(n)

If we instantiate the recursive specification shown in Figure 3-1 with n = 16, our system automatically generates the pattern shown in Figure 3-4.

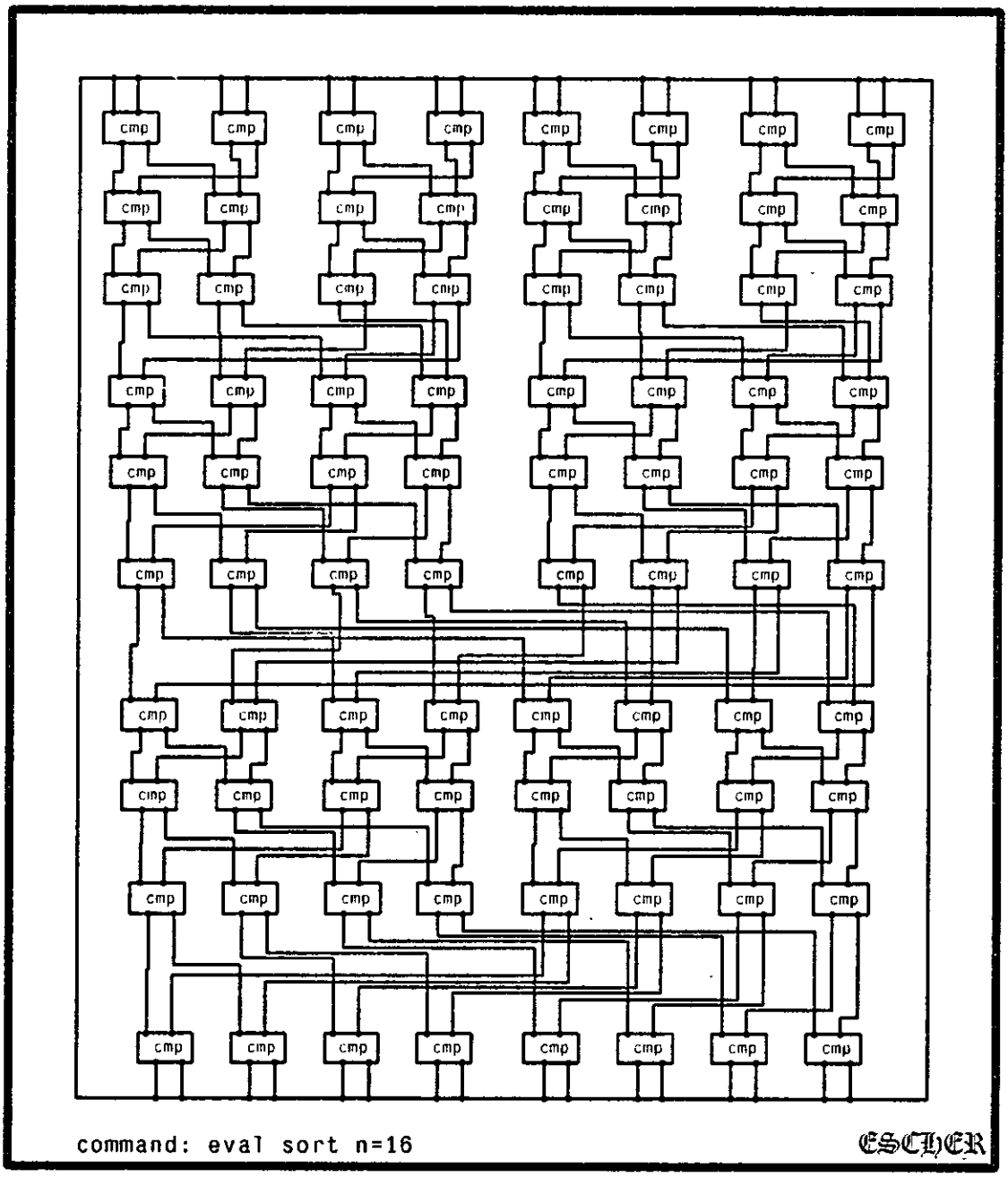

Figure 3-4: Instantiation for SORT(16)

3.2. Example 2: Fast Fourier Transform

The second example is a circuit for computing the Fast Fourier Transform ([4], [11]). Let $\omega = e^{2\pi i/n}$. The *Fast Fourier Transform (FFT)* of $x(0), \ldots, x(n-1)$ is defined for k = 0,1,2, . . . , n-1 by

$$y(k) = \sum_{m=0}^{n-1} \omega^{mk} x(m).$$

This equation can be "folded" to obtain for j = 0,1,2, . . . , n/2-1:

$$y(2j) = \sum_{k=0}^{n/2-1} \omega^{2jk}(x(k) + x(k+n/2)),$$

$$y(2j+1) = \sum_{k=0}^{n/2-1} \omega^{2jk}(\omega^{k}(x(k) - x(k+n/2))).$$

For k = 0,1,2, . . . ,n/2-1, let

$$v(k) = x(k) + x(k+n/2),$$

$$v(k+n/2) = \omega^{k}(x(k) - x(k+n/2)).$$

If we express y in terms of v, we obtain

$$y(2j) = \sum_{k=0}^{n/2-1} (\omega^{2})^{jk} v(k)$$

$$y(2j+1) = \sum_{k=0}^{n/2-1} (\omega^{2})^{jk} v(k+n/2)$$

for j = 0,1,2, . . . , n/2-1. This series of equations can be expressed in matrix form as in Figure 3-5.

$$\begin{bmatrix} y(0) \\ y(2) \\ \cdots \\ \cdots \\ y(n-2) \\ y(1) \\ y(3) \\ \cdots \\ \cdots \\ y(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & w^{2} & w^{4} & \cdots & w^{n-2} & 0 & 0 & 0 & \cdots & 0 \\ & \cdots & & \cdots & & & \cdots & & \cdots & \\ & \cdots & & \cdots & & & \cdots & & \cdots & \\ 1 & w^{*} & w^{*} & \cdots & w^{*} & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 & w^{2} & w^{4} & \cdots & w^{n-2} \\ & \cdots & & \cdots & & & \cdots & & \cdots & \\ & \cdots & & \cdots & & & \cdots & & \cdots & \\ 0 & 0 & 0 & \cdots & 0 & 1 & w^{*} & w^{*} & \cdots & w^{*} \end{bmatrix} \begin{bmatrix} v(0) \\ v(1) \\ \cdots \\ \cdots \\ v(n/2-1) \\ v(n/2) \\ \cdots \\ \cdots \\ \cdots \\ v(n-1) \end{bmatrix}$$

Figure 3-5: Block Diagonal Matrix for Computing FFT

Examination of the block diagonal matrix suggests a recursive circuit for computing the FFT. First, we use a group of *multiplier-adder* cells MAC[0], MAC[1], · · · ,MAC[n/2-1] to transform x[0],x[1], · · · , x[n-1] into v[0],v[1], · · · ,v[n-1]. Each multiplier-adder cell will have two inputs and two outputs. In the case of cell

MAC[k] the two inputs are x[k] and x[k+n/2], and the two outputs are v[k] and v[k+n/2]. In general, the behavior of each MAC[k] cell will depend on the value of k--each must be provided with register holding its particular value of ω^k. For simplicity, however, we will neglect this difference, and assume that MAC is their common source.

We eventually obtain two half size FFT problems: One on v[0], . . .,v[n/2-1] and one on v[n/2], . . .,v[n-1]. The Escher specification of the FFT circuit will contain two recursive instances of FFT(n/2) as shown in Figure 3-6. The cell labeled RPS is just the reverse of the cell PASS defined in the previous example. It takes two sets of n/2 inputs and merges them into n outputs, so that the first set of inputs corresponds to the even-indexed outputs and the second set of inputs to the odd-indexed outputs.

If we instantiate the circuit with n=16, Escher generates the network shown in Figure 3-7. The eight MAC's in the first row have registers holding ω^0, ω^1, ω^2, $\cdots$, ω^7 in sequence from left to right; the eight MAC's in the second row are divided into two groups; the left four in one group and the right four in another. In each group of four, the registers hold ω^0, ω^2,ω^4 and ω^8, respectively. The eight MAC's in the third row are divided into four groups, each of which contains two MAC's, one storing the value ω^0 and one storing the value ω^4. Each MAC in the last row has ω^0 in its register. Although we will not address the problem of initializing the registers, it is not difficult to solve.

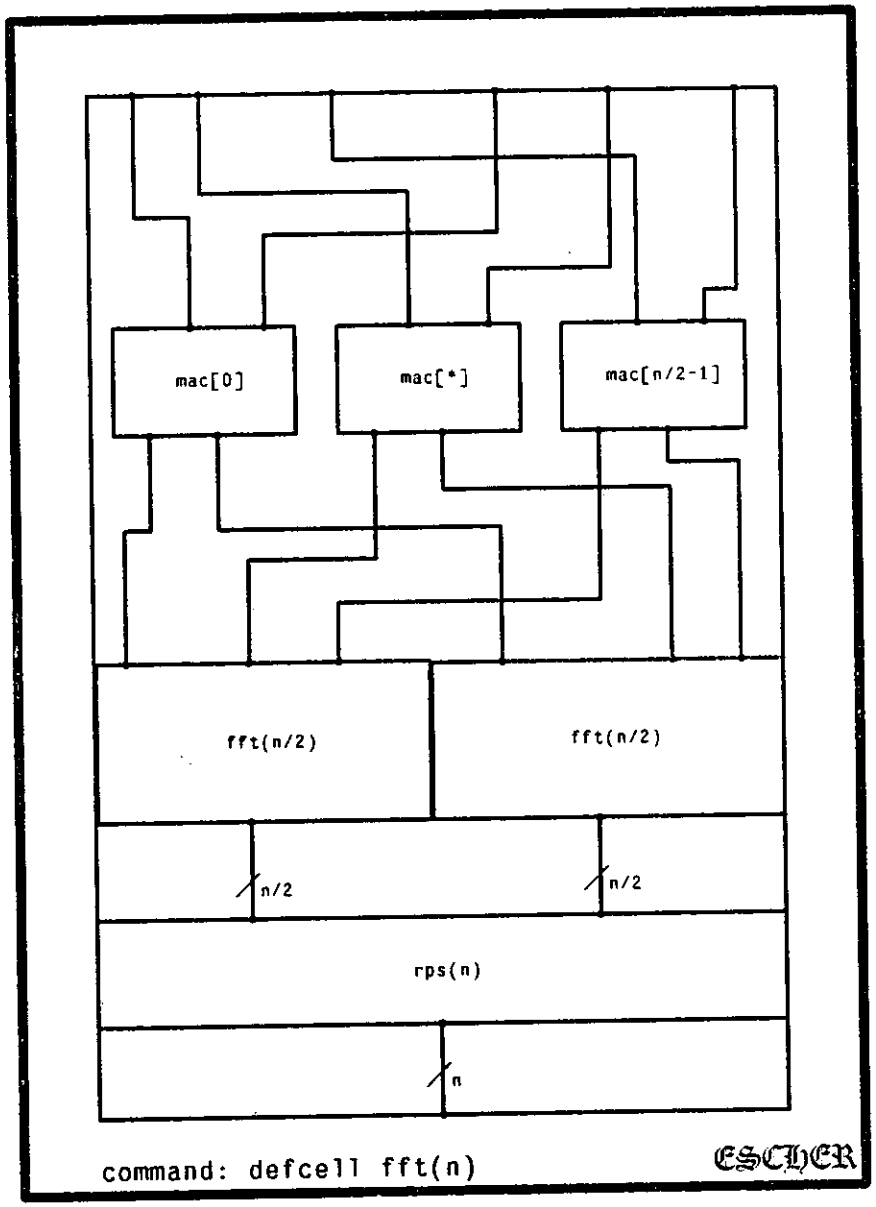

Figure 3-6: Recursive pattern for FFT(n)

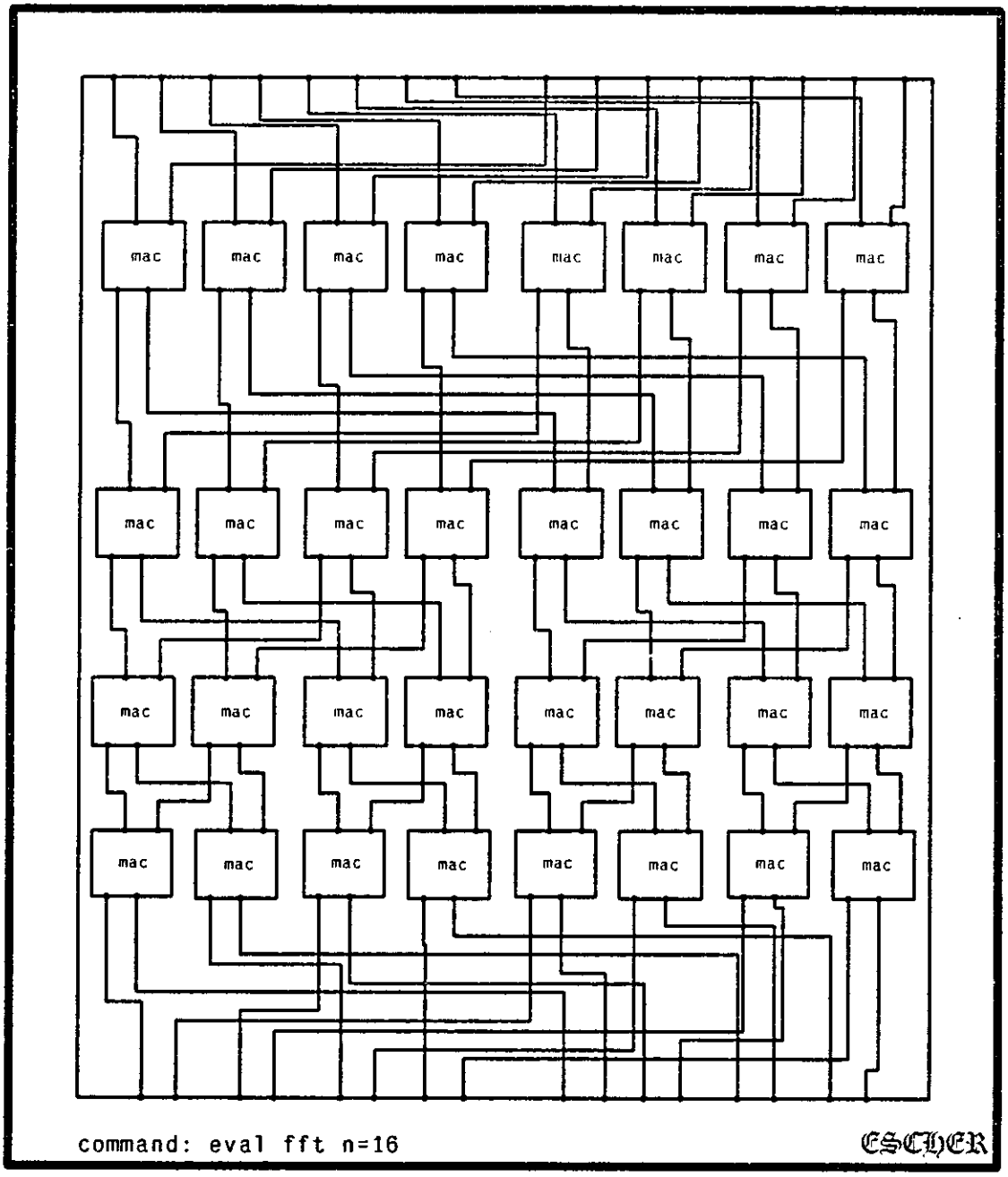

Figure 3-7: Instantiation for FFT(16)

4. A More Complicated Example

In this section we show how to layout the *parallel prefix circuit* described by Fischer and Ladner in [1]. We assume $\otimes$ is an associative binary operator that is implemented by a cell named "OP" with two inputs and one output. The parallel prefix circuit has n inputs and n outputs. The n outputs are the successive *partial products* obtained by using $\otimes$ to combine the inputs. Thus, if the inputs are $x_1, x_2, \ldots, x_n$, then the outputs are $x_1, (x_1 \otimes x_2), \ldots, (((x_1 \otimes x_2) \otimes \ldots) \otimes x_n)$. Figure 4-1 shows the clever recursive circuit suggested by Fischer and Ladner for computing the partial products in parallel.

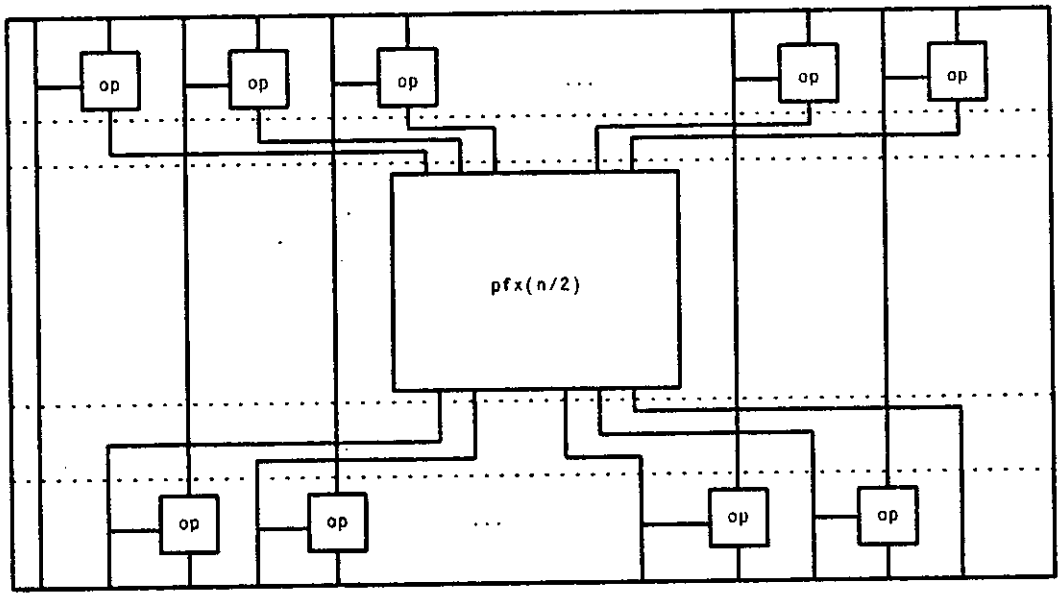

Figure 4-1: Parallel Prefix Circuit

How do we specify the parallel prefix circuit with the Escher system? It is convenient to split the circuit into 5 parts as shown by the dotted lines in Figure 4-1. Each of these parts will correspond to an Escher cell that can easily be defined recursively. See Figure 4-2.

DPASS(n) and UPASS(n) contain only connection wires and are defined in Figure 4-3 and 4-4, respectively. DPART(n) can be split again into two parts, a left part DLEFT and a right part DRIGHT. Each of these parts can, in turn, be defined recursively. See Figures 4-5,4-6 and 4-7. The definition of UPART is similar to that of DPART and will be omitted.

If we instantiate PFX(n) with n=16, Escher will unwind the recursive specification and compact the unwound layout to produce the circuit shown in Figure 4-8.

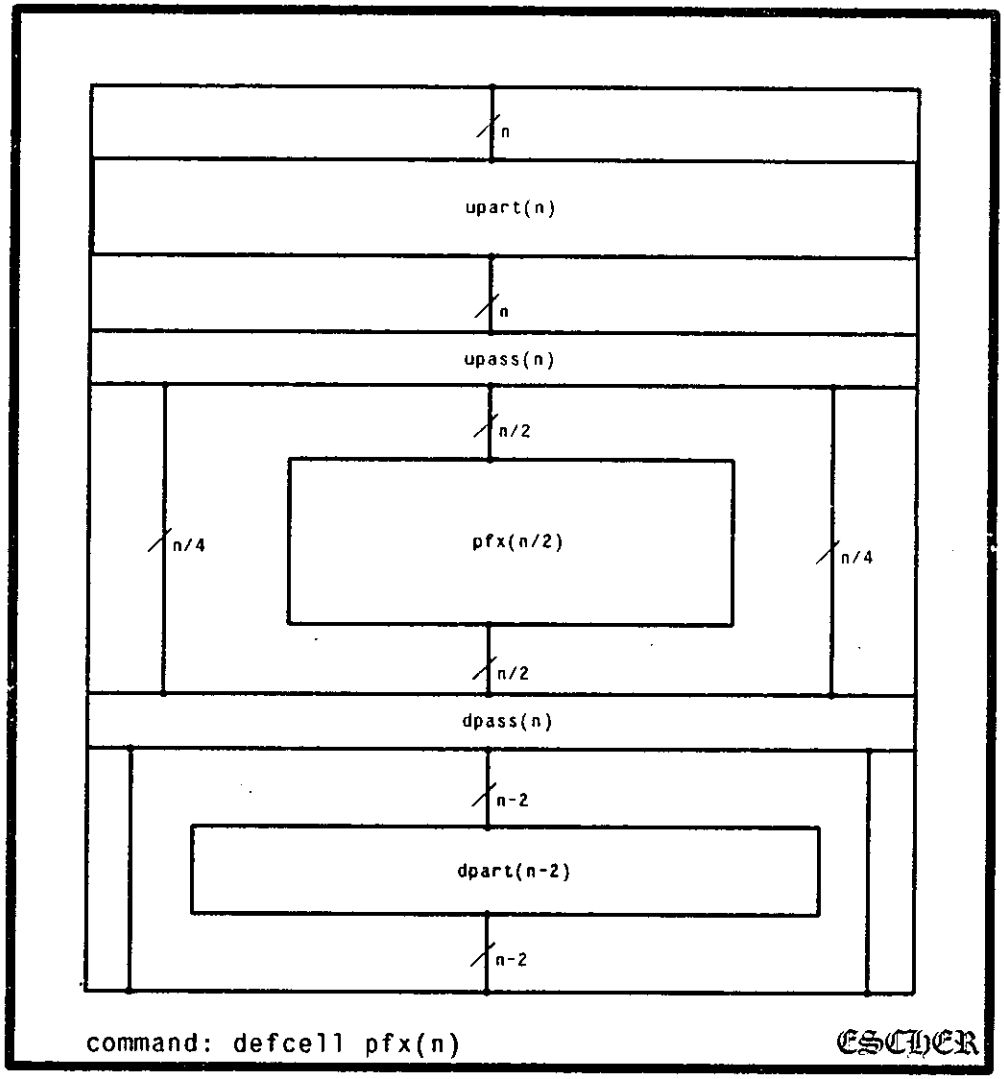

Figure 4-2: Recursive Specification for Parallel Prefix Circuit

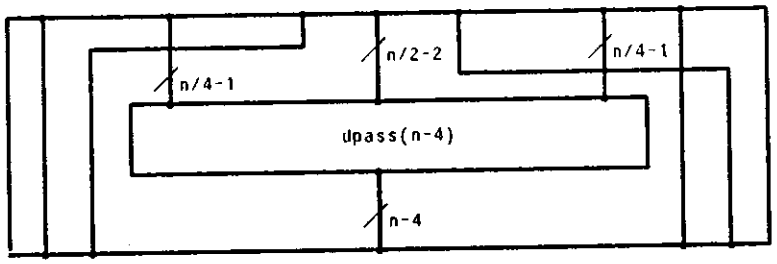

Figure 4-3: Recursive Definition of Subcell DPASS(n)

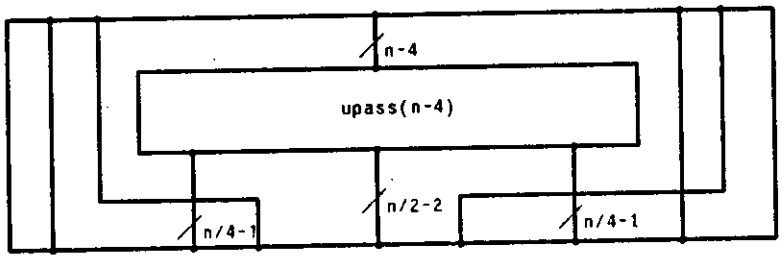

Figure 4-4: Recursive Definition of Subcell UPASS(n)

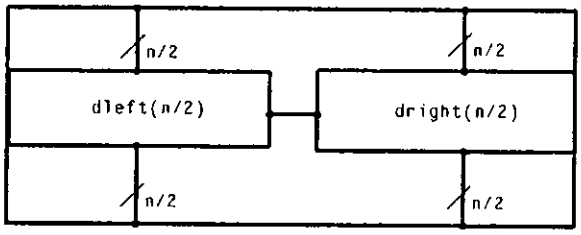

Figure 4-5: Recursive Definition of Subcell DPART(n)

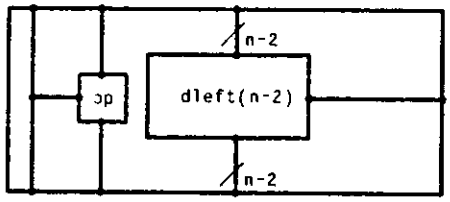

Figure 4-6: Recursive Definition of Subcell DLEFT(n)

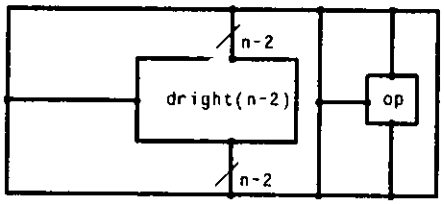

Figure 4-7: Recursive Definition of Subcell DRIGHT(n)

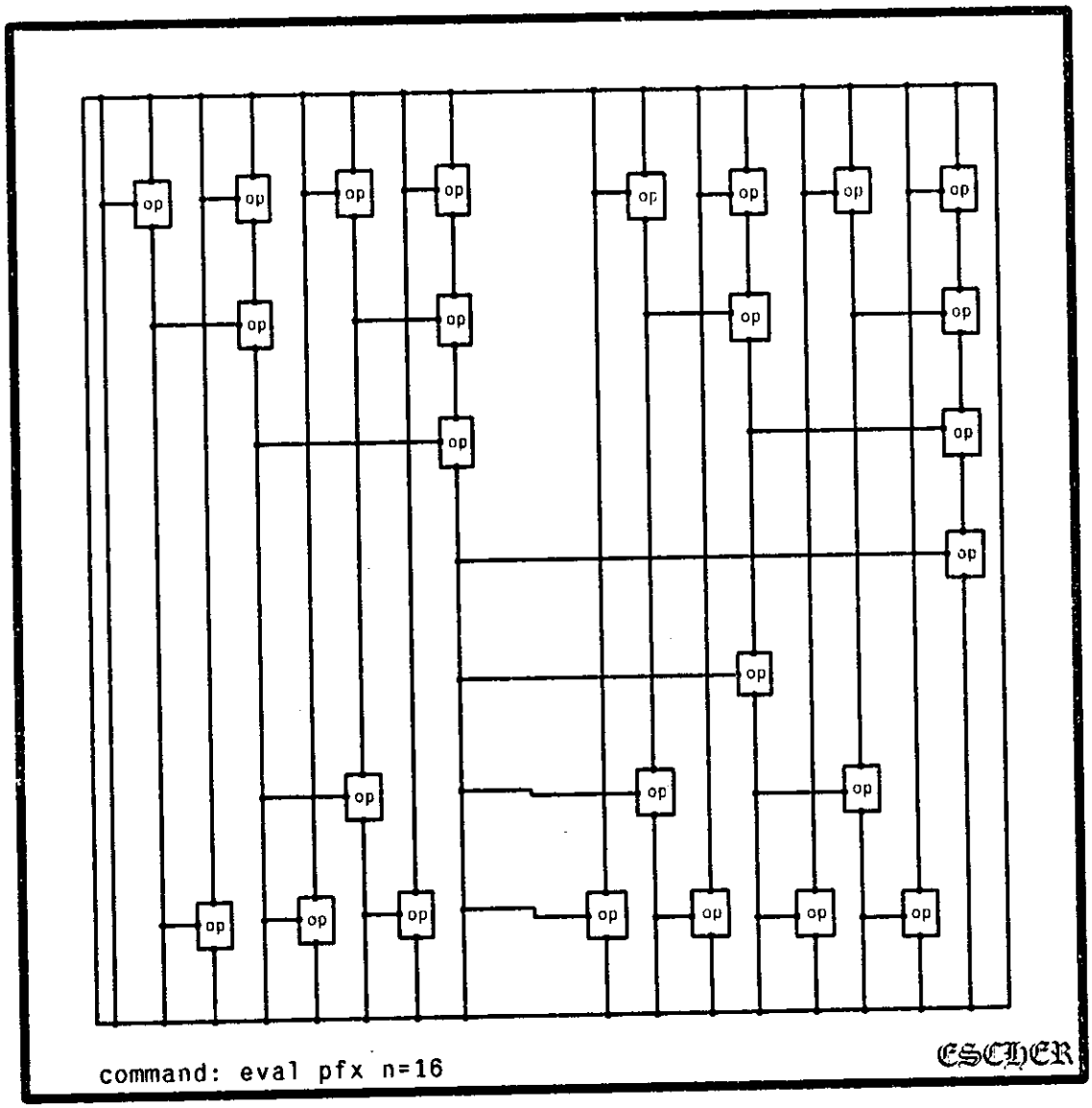

Figure 4-8: PFX(16)

5. Representation of Circuit Components and Structural Elasticity

A cell is represented in the Escher system by a record structure consisting of three fields, the AttributeList, the PointNet, and the SubCellList. The AttributeList contains the name of the cell, its parameter list, and its position (TopY, BottomY, RightX, LeftX) with respect to a fixed coordinate system. The PointNet is used to keep track of the different kinds of points (pins, bends, connectors, vias, transistors, etc.) and their locations. Each point is represented by a record structure that specifies its type, its coordinates PosX and PosY, and how it is connected to the other components of the cell. All of the points in a cell are linked together in an undirected graph structure called the PointNet. From each point in the cell it is possible to find the next connected point in a vertical or horizontal direction by following the appropriate link in the PointNet. The SubCellList contains a descriptor for each component subcell. A subcell descriptor has a pointer to the source of the subcell, an assignment of symbolic expressions for any parameters of the source cell, and information on the position and orientation of the subcell (*i.e.*, whether it has been flipped or rotated). Subcells in a group are linked together in a circular list. Some information in the AttributeList of the source cell, like the cell name, is also duplicated to prevent unnecessary searching.

Only the relative sizes and positions of the various cell components are important. Cells may be expanded or shrunk, points may be moved around, and wires may be lengthened or shortened, provided that the underlying topological structure of the circuit does not change. This structural elasticity is exploited by the Escher system to obtain a good layout and is discussed in more detail below. For simplicity, we initially assume that all subcells, points and wires are at the same level.

We say that subcell SCL1 is *Above* another subcell SCL2, if SCL1.BottomY $\geq$ SCL2.TopY. Similarly, we define the *Below*, *Rightof*, and *Leftof* relations between pairs of subcells. If two subcells are not related by either the *Above* relation or the *Below* relation, they are *Beside* one another. We say SCL1 *Precedes* SCL2 in *position order* if and only if

(SCL1 *Above* SCL2) *Or* ((SCL1 *Beside* SCL2) *And* (SCL1 *Leftof* SCL2)).

In the Tally example in Figure 2-1, each of the subcells MUX[0] ... MUX[n-1] is *Rightof* the subcell tal(n-1) and *Above* the subcell MUX[n]. Similar definitions may be given to describe the relative ordering of pins. In this case, however, the corresponding partial order relations only hold between pins on the same side of a cell.

So far, we have only defined the position order relation between circuit components at the same level. We can extend the relation to apply to components at different levels by requiring that if subcell SCL1 *precedes* SCL2 in position order, then each subcell of SCL1 must precede each subcell of SCL2, etc.

A recursive circuit specification may be unwound into a tree structure in which nodes correspond to cells,

and one node is a son of another if the cell corresponding to the first node is a subcell of the cell corresponding to the second. Thus, a cell will appear at level i in the tree if it is a subcell of a cell that appears at level i-1. A layout is generated from the tree in a *bottom-up* fashion in which layouts are determined for all of the sons of a node before laying out the node itself. To accomplish this task it may be necessary to move various circuit components in order to make room for components generated at lower levels. The algorithms that Escher uses for this purpose are discussed in detail in the next section. To insure correctness we must guarantee that as the program transforms a geometrical pattern, the hierarchical position order among components remains unchanged, even though the absolute size and position of the components may change frequently. We give below some basic rules for deciding when points and subcells may be moved.

- Subcells of a cell may be moved provided that their relative position order remains invariant.
- Each pin on a basic subcell has a fixed position relative to the subcell and can not be moved.
- Pins on some side of a composite subcell may be moved as long as they remain on the same side and their relative order does not change.
- Any non-pin point (*i.e.*, a bend or connector) may be moved, provided all the points whose positions depend on that point are moved accordingly and the move does not violate one of the first three rules.

6. How To Unwind a Recursive Circuit Specification

We begin by describing algorithms for expanding and shrinking cells. There are two instances when this may be necessary: The first occurs when the omitted subcells of a group are filled in. The second instance occurs when a subcell is replaced by a copy of its source. For simplicity, our algorithms for these two cases are only given for the vertical direction. The horizontal direction can be obtained by rotation and need not be given here. When expanding in the vertical direction some parts of the cell must be moved upward, while other parts must be moved downward. However, as long as the guidelines in the previous section are followed, we do not have to worry about changing the behavior of the circuit.

Let Cl(N) be a parameterized cell. Suppose that Cl(V) is the cell obtained from Cl(N) by replacing each expression with its value at N = V. Suppose also that Scl[V1], Scl[*I], Scl[V2] is a subcell group in Cl(V). In order to make room for all of subcells represented by Scl[*I], we must expand (or shrink) Cl(V) as shown in Figure 6-1.

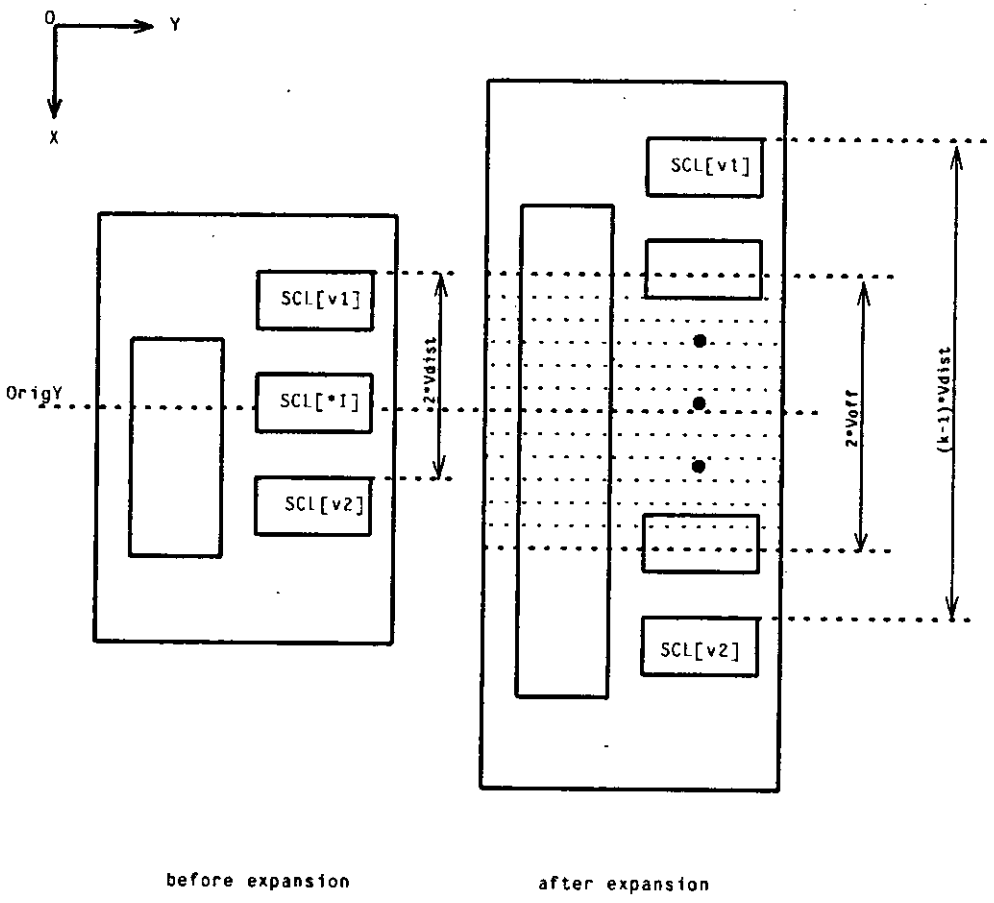

Figure 6-1: Expanding a Group

Vdist is the vertical distance that must be allocated for each subcell in the group, including the space between consecutive subcells. K is the total number of subcells in the group; if K is greater than 3, we must enlarge the space allocated to the group by (K-3) * Vdist units in the vertical direction. Each object in the cell will be moved either up or down according to whether it is above or below the vertical midpoint of the region occupied by the subcell group. (K - 2) new subcells must be created to fill out the group. Points and wires will be added so that each of these subcells has the same set of attachments as Scl[*I].

```
procedure ExpandingGroup;

   K := [(V2-V1) div I] + 1;
   Vdist := (Scl[V1].TopY - Scl[V2].TopY) div 2;
   if K>3 then Voff := [(K-3)*Vdist] div 2 else Voff := 0;
   OrigY := (Scl[V1].TopY + Scl[V2].BottomY) div 2;

   Record all wires and points associated with Scl[*I], then delete
   all of them along with the subcell for Scl[*I];

   if Voff>0 then
        move the part of Cl(V) that is above OrigY up by Voff;
        move the part of Cl(V) that is below OrigY down by Voff;
   endif;

   create (K-2) new subcells that are copies of Scl[*I]; align the
   subcells in the space allocated for the group, making sure that
   the top of successive subcells are separated by Vdist units in
   the vertical direction;

   Connect up the points and wires associated with the individual
   subcells so that each is a copy Scl[*I] in the original diagram;

   endproc;
```

Figure 6-2: Algorithm for Expanding a Group

During the unwinding phase we replace each recursive subcell by an instance of its source cell. For example, when unwinding the TALLY circuit in Figure 2-1 with N = 6 we must first replace the recursive subcell TALLY(5) by a copy of the source cell. When we unwind TALLY(5) we need to replace subcell TALLY(4) by another copy of the source cell. This process continues until a base case is reached.

When we replace a subcell with the body of its source cell, it may be necessary to expand (or shrink) the subcell so that it is the same size as its source. Suppose that Cl is a cell, that Scl is one of its subcells, and that Cll is the source of Scl. Let Voff be half the difference in size in the vertical direction between Scl and its source Cll. When Voff < 0, Scl must be bigger than Cll, so Scl should be shrunk. If Voff > 0, then Scl is smaller than Cll, so Scl should be expanded. When the expansion is made, every object in Cl that is *above* Scl in position order must be moved up by Voff. Likewise, every object in Cl that is *below* Scl must be moved down by Voff. The positions of objects that are *beside* Scl do not need to be changed. See Figure 6-3.

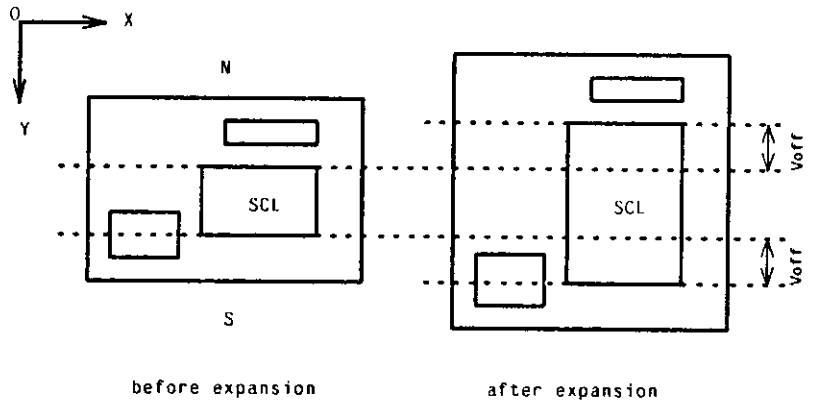

Figure 6-3: Expanding A Subcell

```
procedure ExpandingSubcell;

   Voff := (Cl1.Height - Scl.Height) div 2;

   if Voff<0 then
        for each pin P on Scl's North side,  P.PosY := P.PosY-Voff;
        for each pin P on Scl's South side,  P.PosY := P.PosY+Voff;
   endif;

   if Voff>0 then

      Cl.TopY := Cl.TopY - Voff;
      Cl.BottomY := Cl.BottomY + Voff;

      for each point P,
         if P.PosY<=Scl.TopY then P.PosY := P.PosY - Voff;
         if P.PosY>=Scl.BottomY then P.PosY := P.PosY + Voff;
      endfor

      for each subcell Scl1,

         if (Scl1.TopY<=Scl.TopY) and (Scl1.BottomY>=Scl.BottomY) then
            Cl1.TopY := Scl1.TopY - Voff;
            Scl1.BottomY := Scl1.BottomY + Voff;

         else

            if Scl1.TopY<=Scl.TopY then
               Scl1.TopY := Scl1.TopY - Voff;
               Scl1.BottomY := Scl1.BottomY - Voff;
            endif;

            if Scl1.BottomY>=Scl.BottomY then
               Scl1.TopY := Scl1.TopY + Voff;
               Scl1.BottomY := Scl1.BottomY + Voff;
            endif;

         endif;
      endfor;
   endif;

endproc;
```

Figure 6-4: Algorithm for Expanding a Subcell

Once Scl and its source cell Cll have the same size, we must connect up the pins of Scl. Although corresponding sides of Cll and Scl will have exactly the same number of pins, corresponding pins on the same side will, in general, have different offsets with respect to the center of the side. Let P be a pin on Scl, and let P1 be the corresponding pin on Cll. We assume without loss of generality that both pins are on the South side of their respective cells. Let P2 be that point on the South side of Scl that has the same position with respect to the center of the side that P1 has with respect to the center of the corresponding side of Cll. In general, P and P2 will not coincide and it will be necessary to introduce a *jog* (P,P2) in order to connect them. Frequently, this type of jog can be eliminated by moving some pins or subcells. This problem is addressed by the next algorithm. In order to explain how the algorithm works we consider two cases as illustrated in Figure 6-5. We say that a point U is *rigidly connected* to pin V if U is connected to V by a path consisting of horizontal and vertical wire segments.

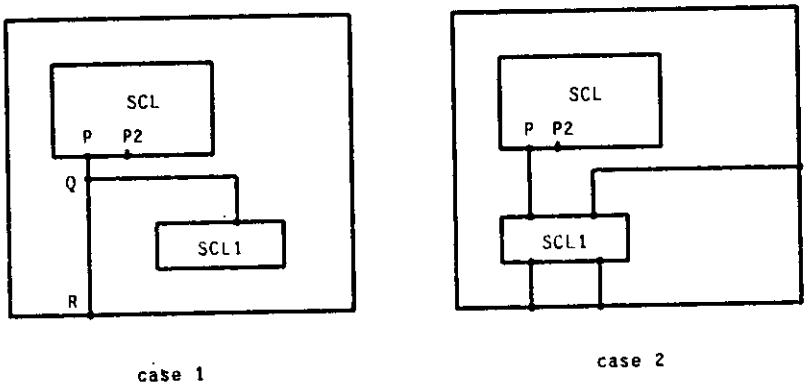

Figure 6-5: Eliminating Jogs (Two Cases).

In the first case, all of the points rigidly connected with P (*i.e.*, points Q and R in Figure 6-5) are movable, so we move all of them right by distance D so that P and P2 coincide. We must be careful not to change the position order among pins on the South side of Scl when we move R. In the second case, it appears that some point Q, rigidly connected with P, is not movable, because it is a pin on the boundary of another subcell Scl1. However, subcell Scl1 and all of the points rigidly connected with its pins are movable. Thus, we can move Scl1 and all of the points rigidly connected to it right by distance D so that P coincides with P2.

```
procedure EliminateJogs;

   for every pin P on the South side of Scl,

      find the corresponding pin P1 on Cl1;

      let C,C1 be the centers for Scl and Cl1;
      NewX := C.PosX + (P1.PosX - C1.PosX);
      NewY := P.PosY;
      add new point P2 at (NewX,NewY);

      D := NewX - P.PosX;

      let Pout be the set of all points rigidly connected to P;

      if every point in Pout is movable, then
         for each P3 in Pout, P3.PosX := P3.PosX+D
         else
         for every subcell Scl1 connected to P,
            let SclOut be the set of all points rigidly connected to Scl1;
            if every point in SclOut is movable, then
               for each P3 in SclOut, P3.PosX := P3.PosX+D;
               Scl1.RightX := Scl1.RightX + D;
               Scl1.LeftX := Scl1.LeftX + D;
            endif;
         endfor;
      endif;

  endfor
endproc;
```

Figure 6-6: Algorithm for Eliminating Jogs.

Finally, we show how to put the previous algorithms together and actually do the unwinding. Our major concern at this point in the algorithm is efficiency. When we unwind a recursive Escher specification, we obtain a tree in which the nodes represent subcells and a directed arc exists between two nodes when the head is a subcell of the tail. We must be careful not to duplicate steps if we encounter the same cell more than once when we traverse the tree. For example, when we instantiate SORT with n=4, we obtain the tree structure shown in Figure 6-7.

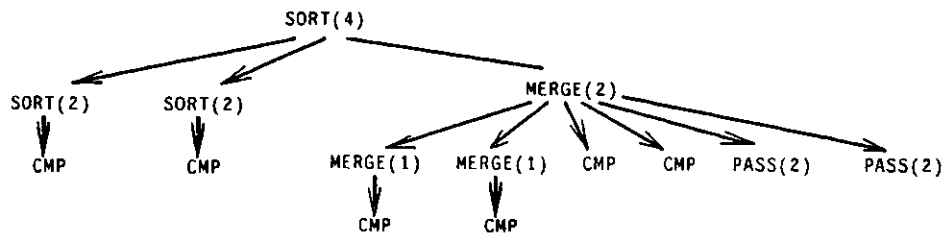

Figure 6-7: Traversal Tree for SORT(4)

In this case there are several duplicates among the 8 terminal nodes and 4 non-terminal nodes. In order to unwind SORT(4), we have to unwind SORT(2) twice and MERGE(2) once; when we unwind MERGE(2), we must unwind PASS(2) twice, MERGE(1) twice, and CMP twice. In fact, if this representation is used, it is possible to create examples in which the number of duplicated steps will be exponential in the size of the original Escher specification.

Instead, Escher uses a directed acyclic graph structure to represent the nesting of subcells. We call this data structure the *Subcell Nesting Graph or SNG*. Since each subcell corresponds to at most one node in the SNG, it is only necessary to unwind a given subcell once. The graph for SORT(4) is shown in Figure 6-8. Note that each of the subcells SORT(4), SORT(2), CMP, MERGE(2), MERGE(1), and PASS(2) is represented uniquely this time.

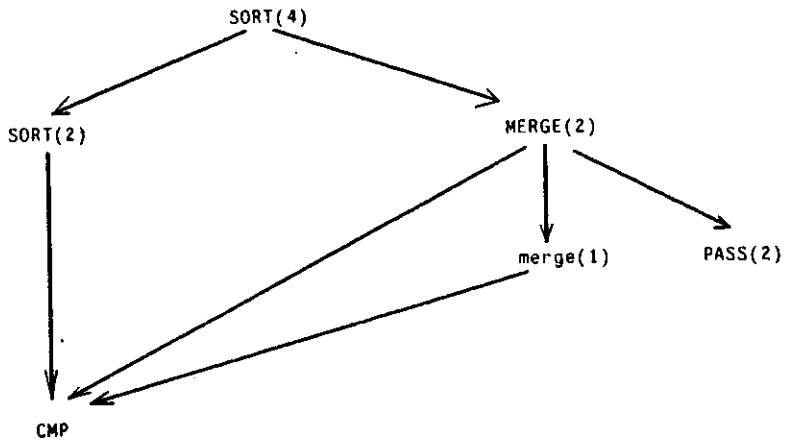

Figure 6-8: Directed Acyclic Graph for SORT(4)

The unwinding algorithm consists of two phases. In the first phase we evaluate all of those expressions that depend on the parameters of the cell and create the SNG. Expressions may appear in the specifications of groups and buses, and they may be used as parameters of lower level subcells. After we have figured out the exact number of subcells in a subcell group, we use the algorithm in Figure 6-2 to obtain enough space for the omitted subcells in the group; then we copy the subcells into the cell. After a cell has been evaluated it will be linked to its source cell in the SNG. The SNG for cell CL(V) will not be complete until all of its descendant subcells have been processed in this manner.

The second phase in the unwinding process is a depth first traversal of the SNG. When all of the subcells of a cell in the SNG have been unwound, we replace each subcell with its source body and mark the cell as unwound. The algorithm in Figure 6-4 is used to obtain enough space for filling in the subcell bodies. The algorithm in Figure 6-6 is used for eliminating jogs in wires that result from these substitutions.

```
procedure Eval(Cl,OrigCl, N, V);

   name N;

   if Cl(V) is already in the SNG  then return
   elseif Cl is a basic cell then add a Cl-node into SNG
   else

      evaluate all of the expressions in Cl(N), replacing N by its
      value V;
      expand all subcell groups using the algorithm in Figure 6-2;
      replace each bus by a number of wires equal to the bus width;

      for each subcell Scl1 do
         let the source of Scl1 be Cl1(V1);
         Eval(Cl1,Cl,N,V1);
      endfor;

      add a Cl(v)-node into SNG;

   endif;

   set link from OrigCl to Cl(V);

endproc;
```

Figure 6-9: Algorithm for constructing Subcell Nesting Graph.

```
procedure Unwind(Cl(v));

   for each descendant Cl1(v1) of Cl(v) do
      if Cl1(v1) is not unwound then Unwind(Cl1(v1));
   endfor;

   for each subcell Scl of Cl(v),
      expand Scl to be the same size as its source;
      map the pins of Cl1 onto Scl and minimize the number of jogs using
      the procedure described in Figure 6-6;
      copy Cl1(v1) into Scl;
   endfor;

   mark Cl(v) as unwound;

endproc;
```

Figure 6-10: Algorithm for Unwinding Recursive Cell Specifications).

Finally, some simple compaction algorithms are used to shorten wires and move subcells closer together. It should be noted that these algorithms may violate the position order relation among subcells that is described in section 5. For example, the compaction algorithms were used to obtain Figure 4-8. The position order among subcell OP's is not the same as that given in the original specification of the parallel prefix circuit.

7. Conclusion and Directions for Future Research

We believe that ultimately recursion will play much the same role in hardware design that it has in software design. Although recursion has always been an indispensible tool for theoretical investigations in algorithm design, only in the last few years has it become respectable to write application programs that use recursive procedures. The acceptance of recursion is a result of two factors: First, many software designers have come to realize that it natural to express certain algorithms recursively-- particularly those that access recursive data structures. Secondly, advances in computer architecture, like hardware stacks and displays, have decreased the overhead associated with recursive procedure calls. We believe that an analogous process will occur in hardware design. When design environments routinely provide support for recursion, designers will begin to find elegant recursive solutions for problems that they currently must solve in an awkward manner using iteration alone. Since recursive hardware designs are implemented by unwinding the recursion, the overhead in efficiency that is associated with the use of recursion in software will not be a problem. We further believe that our use of parameterized subcell and group specifications will be of practical importance in any completely general graphical design system, even if full recursion is not supported.

Finally, we list below some of the problems with the current system that we hope to address in a future version:

- **Multiple parameters.** As currently implemented, the Escher system only permits cell specifications with a single recursive parameter. A number of interesting examples can be specified most naturally by using multiple recursive parameters. It should be fairly easy to modify the current implementation so that multiple parameters are permitted.

- **Compaction and optimization.** The layouts produced by our system frequently contain long wires and have area that grows more rapidly with the recursion depth than necessary. Although we have implemented some simple compaction algorithms, we believe that this problem requires much more thought. It may be possible to design compaction algorithms that take advantage of the hierarchical structure of Escher specifications. However, the simple algorithms that have already been implemented do not make use of this information.

- **Combined textual and geometric description.** For certain applications like simulation a textual circuit description may be quite useful. We envision a VLSI design system with multiple *windows* which would permit *both* textual and geometric descriptions of circuit components. One window would contain a geometrical representation of the circuit like the one described in this paper. Another window would contain a representation of the circuit in an appropriate (textual) hardware description language. The textual description could be used directly for simulation,

verification, etc. A change in the geometrical description would be automatically reflected by a corresponding change in the HDL representation. The dual representation would provide access to the best features of both types of design systems.

REFERENCES

1. M. Fischer and R. Ladner. "Paralllel prefix computation". *Journal of the ACM 27*, 4 (1980).

2. S.M.German, K.J.Lieberherr. "Zeus: a language for expressing algorithms in hardware". *Computers* (1985).

3. J.Ousterhout. "Caesar: An interactive layout editor for VLSI design". *VLSI design* (Fourth Quarter 1981), 34-38.

4. L Johnson, U. Weiser et tal. Towards a formal treatment of VLSI arrays. Caltech conference on VLSI, January, 1981, pp. 375-398.

5. D.E.Knuth. *The art of computer programming.* Volume : *Sorting and searching.* Addison-Wesley, 1973.

6. R.J.Lipton, S.C.North, R.Sedgewick et tal. ALI: a procedural language to describe VLSI layouts. 19th design automation conference, IEEE, 1982, pp. 467-474.

7. W.K.Luk, J.E.Vuillemin. Recursive implementation of optimal time VLSI integer multipliers. VLSI design of digital systems, ed. F.Anceau & E.J.Aas, 1983, pp. 155-168.

8. C.A..Mead, L.A.Conway. *Introduction to VLSI systems.* Addison-Wesley, 1980.

9. Mary Sheeran. muFP-- An algebraic VLSI design language. PRG-39, Oxford University Computing Lab., November, 1984.

10. P.Henderson. Functional geometry. Symposium on LISP and functional programming, ACM, 1982, pp. 179-187.

11. C.D.Thompson. "Fourier transforms in VLSI". *IEEE transaction on computers c32*, 11 (1983), 1047-1057.